Supervisor Runtime

Tera Computer Company
2815 Eastlake Ave E
Seattle, WA 98102

August 9, 1993

1 Introduction

There are two types of chores that execute in the supervisor: supervisor-promoted chores (`SPChores`) are created as a result of a user chore entering the OS via a call to a supervisor library routine, and execute within the user's address space; supervisor daemons (`SvDaemons`) exist independently of any particular user task, and execute within the OS domain on each processor. SPChores execute on behalf of the user to fulfill a particular request for system services, whereas SvDaemons provide assistance for system-wide services such as networking and file systems. The runtime model for SPChores has been defined elsewhere. This document presents a design for the SvDaemon runtime model and interface. This is a preliminary design, and subject to change.

2 Motivation

During normal execution, the Tera Operating System generates a large volume of work that must be processed in parallel in order to provide reasonable efficiency, throughput, and scalability. Much of this parallelism can be achieved by having the originating SPChore do most of the work. However, there are occasions in which there is no SPChore associated with the generated work. For instance, the networking subsystem needs to process outgoing and incoming network packets in parallel. Outgoing packets are processed in parallel by the SPChores that are sending the packets. However, when an incoming packet arrives, the ultimate task/chore destined to receive the packet cannot be determined until the TCP header of an inbound packet is analyzed. Normally, these are handled via software interrupts in Unix. In the Tera OS, they are serviced in parallel by SvDaemons. In the following discussion, units of work that are executed in parallel are referred to as chores.

3 Requirements

The following list summarizes the characteristics of OS chores:

- Chores are prioritized.
- Chores may block during execution.
- The producer does not require notification of completion.

- There are a relatively small number of chore types.

Each chore has a priority associated with it. The system ensures that a chore of priority j will be chosen for execution prior to any chore of priority $i < j$. Note that this does not ensure that the higher priority chore will be executed first, since SvDaemons execute in parallel. Chores of the same priority are chosen in FIFO order.

Chores may block during execution. If a chore blocks, the SvDaemon servicing the chore will be released to seek additional work.

Once a chore begins execution, it cannot be interrupted. This brings up a possible problem. In Unix, when a higher priority event occurs, it is handled immediately (provided it is not masked), even in the midst of an interrupt handler. In the Tera supervisor runtime, what happens if all SvDaemons are busy and a high priority event occurs? One possible solution is to force the creation of an additional SvDaemon (subject to a maximum).

In general, the only agent that may be interested in the completion of the chore is an SPChore that is sleeping, waiting for some event to occur. For instance, in networking, an SPChore may be sleeping, waiting for a packet to arrive. In such cases, when the chore completes execution, it notifies the waiter via the normal Unix wakeup() call. The producer that originally generated the chore is never interested in its completion [1].

Currently, we have identified only a handful of chore types:

1. process incoming network packets
2. process incoming Hippi client messages
3. NFS client
4. NFS server

In addition, if we decide to implement Unix streams, the handling of the messages in the service queues would be handled via SvDaemons. Note that there can be multiple instances of each chore type.

4 Basic Execution Model

This section describes the basic execution model for SvDaemons. One protection domain on each processor is reserved for use by the operating system, and is referred to as the OS domain. At boot time, one or more supervisor daemons are created in each OS domain. Each daemon executes as a

[1] This differs from the user-level runtime model, in which the producer is often interested in the completion of the chore, and waits for notification via the future mechanism.

virtual processor. The pseudocode below illustrates the body of the virtual processor loop.

```
vploop() {
    forever()
    {
        if (myteam->unblockedQueue not empty)
        {
            ccb = myteam->unblockedQueue.dequeue()
        }
        else if (workQueue not empty)
        {
            continuation = workQueue.dequeue()
            allocate ccb
            bind continuation to ccb
        }
        execute chore
        if (chore completed execution)
            free ccb
    }
}
```

When an executing chore becomes blocked, its chore control block is placed on a per-team blocked list. When the chore becomes unblocked, it is moved to a per-team unblocked queue (prioritized). The unblocked queue is examined prior to the global workQueue, giving preference to unblocked chores over new chores [2]

The workQueue is a logical queue containing a prioritized list of continuations [3]. A continuation can be thought of as an infant chore; when it is bound to a chore control block, it matures into a chore. The continuation contains at least a pointer to a function, and arguments that are passed to the function. The continuations are dynamically generated by executing SPChores and/or SvDaemons. The continuation must be bound to a chore control block prior to execution; the chore control block contains a stack, save areas for trap handling, and various other information. When the continuation is bound to the chore control block, the function arguments are transferred from the continuation to the registers normally used for passing parameters, according to compiler convention (maximum of eight arguments allowed). Currently, the continuation is then freed when the chore completes execution.

When the chore completes execution, the daemon returns to the virtual processor loop. Similarly, if the chore becomes blocked, it is placed on a blocked list, and the daemon returns to the virtual processor loop. If the chore completed execution, the chore control block is freed [4].

---

[2]Note that this may violate the priority order, since chores in the workQueue may have higher priority than those in the unblocked queue. If this is a problem, the unblocked queue will need to be prioritized as well, and the vploop will need to compare the priorities.

[3]Note that the actual implementation will likely use multiple queues to emulate a single logical queue.

[4]An optimization is to momentarily hold on to the chore control block, in case it can be reused in the next iteration of the virtual processor loop.

5 Growth Policy

Unlike the user runtime, the number of SvDaemon teams is statically determined at boot time, and is equal to the number of processors. However, the number of SvDaemons actively executing on each processor varies according to the amount of work there is to do.

The goals of the growth policy are:

- process work orders in a timely fashion
- do not deprive user-level tasks of necessary resources
- do not react to changes in the work load too quickly
- balance the load across processors

The processing capacity must be such that SvDaemons do not become a bottleneck for OS services. On the other hand, the number of daemons should not be so high as to cause a negative impact on the parallelism of user-level tasks. Decisions to add or retire an SvDaemon should be made on the basis of medium to long term changes in the work load, rather than brief and transitory changes [5]. It is also desirable to balance the load across the processors (how do we do this?).

The basic metric for growth decisions is the total size of the work queue. Beyond this, details of the growth policy have not been defined yet. The intention is to start with a simple policy, and to optimize the policy based on observations on the actual system. Here are some initial thoughts, inspired by the user runtime growth policies. The basic questions are:

- deciding if growth is needed
- deciding where to add a vp, and how many to add
- deciding when to retire a vp
- how often is the growth/shrink policy evaluated

To decide whether growth is needed, consider the amount of pending work per virtual processor, and add a vp if the ratio is high; e.g. if the ratio of the sum of the sizes of the unblocked queues and the global work queue, to the total number of SvDaemons is greater than m (e.g. $m = 5$), create a new daemon. Evaluate the growth algorithm periodically; e.g. every k-th insertion into the global workQueue, and/or the blocked or unblocked queues.

Add the vp to the team with the least number of vps. An optimization is to examine only the next i (e.g. $i = 3$) teams, rather than searching the entire team list, or maintaining an ordered list.

To decide whether to retire A simple algorithm should suffice for deciding when to retire. If both the unblocked queue and the workQueue are empty for n (e.g. $n = 10$) iterations of the virtual processor loop, the daemon voluntarily retires.

Once the decision has been made to create a daemon on a particular processor, how is the daemon created? One way might be to send a request to the KernelDaemonTeam residing on the topic

[5] Need to consult user runtime to see how it manages this.

processor. An alternative is to craft a chore control block and enqueue it with highest priority on the unblocked queue of the SvDaemonTeam residing on the topic processor. This will work provided there is always at least one SvDaemon executing on a processor.

Possible optimizations:

- Each team has a maximum allowed number of vps.
- Add n vps at a time instead of one, where n is the required number necessary to make the desired work-per-vp ratio exist. This should help mitigate the problem of the runtime growing too slowly.
- Add an interface to allow the client to demand a growth assessment (e.g. if the client is generating lots of work).

6 Interface

The interface consists of the following routine:

```
void choreCreate(PriorityLevel_k priority,
                 void (*function)(),
                 int numArgs,
                 ... /* argument list */);
```

This routine arranges for the specified function to be called with the given argument list sometime in the future. It allocates a continuation and enqueues it with the specified priority on the global work queue. The practical motivation for using an explicit function call rather than the *futures* abstraction provided by the compiler is mainly to allow the same client code to execute under both Awesime and zebra (otherwise, the client code would be littered with `#ifdef` directives). Also, the supervisor runtime is much simpler than the user runtime, and does not require all the functionality provided by futures.

7 Implementation Issues

The pseudocode below illustrates how the `choreCreate` routine might be implemented. For compatibility, it uses the compiler defined data type for continuations (in case future implementations

want to take better advantage of the compiler facility).

```
void choreCreate(priority, func, numArgs, ...)
{
    assert(numArgs <= MAX_ARGS)
    Continuation c = continuation_alloc(numArgs);
    c->priority = priority;
    c->func = func;
    for (i = 0; i < numArgs; i++)
        c->args[i] = argi;
    workQueue.enqueue(c);
}
```

The routine first allocates a continuation structure, fills it in with arguments, then enqueues it on the global work queue. The implementation uses `varargs` to parse the argument list. The maximum number of arguments allowed is limited to a predefined constant (currently four). This allows the continuation structures to be fixed size. A cache of free continuation structures is maintained.

The work queue implementation needs to support a high degree of concurrent access. A possibility is a queue of queues, with a priority associated with each queue. The individual queues can then be simple first-in-first-out queues.

Since SvDaemons share the OS domain with KernelDaemons, they must cooperate with the KernelDaemon policy for stream management (to maintain a minimum stream reservation).